Evaluating Static Analysis Defect Warnings On Production Software

Nathaniel Ayewah, William Pugh
University of Maryland
ayewah,pugh@cs.umd.edu

J. David Morgenthaler, John Penix, YuQian Zhou
Google, Inc.
jdm,jpenix,zhou@google.com

Abstract

Static analysis tools for software defect detection are becoming widely used in practice. However, there is little public information regarding the experimental evaluation of the accuracy and value of the warnings these tools report. In this paper, we discuss the warnings found by FindBugs, a static analysis tool that finds defects in Java programs. We discuss the kinds of warnings generated and the classification of warnings into false positives, trivial bugs and serious bugs. We also provide some insight into why static analysis tools often detect true but trivial bugs, and some information about defect warnings across the development lifetime of software release. We report data on the defect warnings in Sun's Java 6 JRE, in Sun's Glassfish JEE server, and in portions of Google's Java codebase. Finally, we report on some experiences from incorporating static analysis into the software development process at Google.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.2.4 [*Software/Program Verification*]: Reliability

General Terms Experimentation, Reliability, Security

Keywords FindBugs, static analysis, bugs, software defects, bug patterns, false positives, Java, software quality

1. Introduction

Static analysis for software defect detection has become a popular topic, and there are a number of commercial, open source and research tools that perform this analysis. Unfortunately, there is little public information about the experimental evaluation of these tools with regards to the accuracy and seriousness of the warnings they report. Commercial tools are very expensive and generally come with license agreements that forbid the publication of any experimental or evaluative data.

Even when data is published, there is not significant agreement on standards for how to evaluate and categorize warnings. Some warnings reflect situations that cannot occur and that a smarter tool might be able to show are infeasible. But many other warnings, while reflecting true defects in the code, do not significantly impair the intended functionality of the application. Should both of these be counted as false positives, or only the first? The paper by Wagner et al [12] regards a defect warning as a false positive unless the developer believes that the defect could result in significant misbehavior of the application; however, this classification was not clear in the paper and only confirmed through personal communication. Clearly, clarification on this issue is necessary when discussing false positives from static analysis.

Since software rarely comes with specifications that can be interpreted by automatic tools, defect detection tools don't try to find places where software violates its specification. Instead, most defect detection tools find inconsistent [6] or deviant [5] code: code that is logically inconsistent, nonsensical, or unusual. But inconsistent or nonsensical code, while undesirable, may not actually result in the application behaving in a way that substantially deviates from its intended functionality. We discuss this situation and give examples of such defects in Section 3. This issue is particularly exacerbated in a memory-safe language such as Java. In a memory-unsafe language such as C, there is no such thing as a harmless (but feasible) null pointer dereference or buffer overflow. But in Java, dereferencing a null pointer and throwing a null pointer exception may be the most appropriate response when a null value is unexpectedly passed as a method parameter.

There are many contexts for using static analysis for defect detection. One context is when performing code review of a newly written module. In such cases, developers will likely be interested in reviewing any warnings of questionable code, and would likely want to correct confusing or misleading code, even if the confusing or misleading code doesn't result in misbehavior. Another context is when you wish to look for defects in a large existing code base. In this context, the threshold for changing code in response to a warning is higher, since the issue has been in production without causing serious apparent problems, and you would need to get a developer to switch from other tasks and gain familiarity with the code in question before they could make a change they would be confident in. In this paper, we largely concern ourselves with the second context: applying static analysis to existing code bases.

In this paper, we report on results of running FindBugs against several large software projects, including Sun's JDK (Section 4), Sun's Glassfish J2EE server (Section 5) and portions of Google's Java code base (Section 6).

2. FindBugs

FindBugs is an open source static analysis tool that analyzes Java classfiles looking for programming defects. The analysis engine reports nearly 300 different bug patterns. FindBugs has a plugin architecture, in which detectors can be defined, each of which may report several different bug patterns. Rather than use a pattern language for describing bugs (as done in PMD[4] and Metal[3]), FindBugs detectors are simply written in Java, using a variety of techniques. Many simple detectors use a visitor pattern over the classfiles and/or the method bytecodes, often using a state machine and/or information about the types, constant values, and special

PASTE'07 June 13–14, 2007, San Diego, California, USA.

flags (e.g., is this value the result of calling hashCode) about values stored on the stack or in local variables. But detectors can also traverse the control flow graph, using the results of data flow analysis such as type information, constant values and nullness. The data flow algorithms all generally use information from conditional tests, so that information from `instanceof` tests and null tests are incorporated into the analysis results.

FindBugs does not perform interprocedural context sensitive analysis. However, many detectors make use of global information such as subtype relationships and which fields are accessed across the entire application. A few detectors use interprocedural summary information, such as which method parameters are always dereferenced.

Each bug pattern is grouped into a category (e.g., correctness, bad practice, performance and internationalization), and each report of a bug pattern is assigned a priority, high, medium or low. The priorities are determined by heuristics unique to each detector/pattern, and are not necessarily comparable across bug patterns. In normal operation, FindBugs does not report low priority warnings. During the first summer FindBugs was applied at Google, substantial effort was put into trying to ensure that issues reported as high or medium priority correctness issues were issues that had a low false positive rate, and that developers would be interested in examining all issues reported as high or medium priority correctness issues, even in large code bases.

Perhaps the most important aspect of FindBugs is how new bug detectors are developed: by starting with real bugs, and developing the simplest possible technique that effectively finds those bugs. This approach often allows us to go from finding a particular instance of a bug to implementing a detector that can effectively find it in a matter of hours. Many bugs are really very simple; one of the bug patterns most recently added to FindBugs is to cast an integer value to a `char` and then check to see if the result is `-1`. This bug was inspired by a post on `http://worsethanfailure.com/`, and within less than an hour we had implemented a detector that found 11 such errors in Eclipse 3.3M6.

FindBugs can be run from the command line, Ant or Maven, or in a GUI, Eclipse or NetBeans. The analysis results can be saved in XML, which can then be further filtered, transformed, or imported into a database. FindBugs supports two different mechanisms for matching defect warnings from different analysis runs (trying to match up a defect warning in one analysis with the corresponding warning in another analysis, despite program changes that may have changed line numbers) [10].

2.1 Deploying FindBugs at Google

At Google, FindBugs is automatically run over any modified code, generating XML. The XML file is then imported into a database, which also contains reports from other static analysis tools. The instance-hash mechanism described in [10] is used to allow a defect to be matched with previous reportings of that warning. At the moment, the database is reviewed by two engineers, who perform bug triage, evaluating which warnings should be reported to developers. A cost/benefits analysis showed that this would be the most effective way to evaluate which bug patterns were consistently appropriate to report to developers and to gain more experience with capturing and reporting FindBugs results. As experience and confidence with FindBugs grows, Google is working to incorporate FindBugs warnings more directly into the developer workflow.

3. True But Low Impact Defects

One unexpected finding from looking at a number of real defect warnings is that there are a number of cases in which FindBugs has correctly diagnosed what seems to be an obvious defect, yet it is also clear that the defect will not result in measurable misbehavior of the program. In fact, sometimes the misbehavior is intentional.

```
// com.sun.jndi.dns.DnsName, lines 345-347
 if (n instanceof CompositeName) {
    n = (DnsName) n;   // force ClassCastException
 }

// com.sun.java.util.jar.pack.Attribute,
// lines 1042-1043
if (layout.charAt(i++) != '[')
    layout.charAt(-i);  // throw error
```

Figure 1. Two intentional errors

```
// sun.jdbc.odbc.JdbcOdbcObject, lines 85-91
if ((b[offset] < 32) || (b[offset] > 128)) {
  asciiLine += ".";
}
else {
  asciiLine += new String (b, offset, 1);
}
```

Figure 2. Masked Error

3.1 Deliberate errors

Figure 1 shows two examples of code containing deliberate errors (code that is intended to result in a RuntimeException being thrown). The developer apparently believed this was preferable to explicitly creating and throwing an exception.

3.2 Masked errors

Figure 2 shows a masked error. The variable `b` is a byte array. Any value loaded from a byte array is treated as a signed byte and sign extended to an integer in the range -128 to 127. Thus, the test `b[offset] > 128` will always be false. However, the cases where the value would be greater than 128 if treated as an unsigned byte are caught by the test `b[offset] < 32`, so the defect cannot actually cause misbehavior. This example also shows a fairly common phenomenon where warnings are closely associated with other questionable code. Here, the code is constructing single character strings and incrementally appending them to a String (which has quadratic complexity) rather than simply appending a character to a StringBuffer.

3.3 Infeasible statement, branch, or situation

Some errors only arise in a situation that the developer believes to be impossible but cannot easily infer by direct examination of the code. Figure 3 shows an example of an infeasible error: the developer clearly believes that the exceptions cannot arise in this situation. In this example, interpreting the comments would be worthwhile, but the developers beliefs are not always this well documented.

One appropriate remedy for an infeasible error would be to fail an assertion. However, the case for modifying existing code to add an assertion for a situation believed to be infeasible is weak.

3.4 Already Doomed

Sometimes, an error can occur only in a situation where the computation is already doomed, and throwing a runtime exception is not significantly worse than any other behavior that might result from fixing the defect. Figures 4 – 6 shows three examples of doomed situations. In Figure 4 the comment indicates the intention of the

```
// com.sun.corba.se.impl.dynamicany.DynAnyComplexImpl
String expectedMemberName = null;
try {
    expectedMemberName = expectedTypeCode.member_name(i);
} catch (BadKind badKind) { // impossible
} catch (Bounds bounds) { // impossible
}
if ( ! (expectedMemberName.equals(memberName) ... )) {
```

Figure 3. Infeasible situation

```
// com.sun.org.apache.xml.internal.security
//    .encryption.XMLCiper
// lines 2224-2228
if (null == element) {
  //complain
}
String algorithm = element.getAttributeNS(...);
```

Figure 4. Doomed situations: vacuous complaint

```
// com.sun.org.apache.xalan.internal
//        .xsltc.runtime.output
//        .TransletOutputHandlerFactory
// lines 146-174 (slightly modified for brevity)
SerializationHandler result = null;
if (_method == null)
    result = new ToUnknownStream();
else if (_method.equalsIgnoreCase("xml"))
    result = new ToXMLStream();
else if (_method.equalsIgnoreCase("html"))
    result = new ToHTMLStream();
else if (_method.equalsIgnoreCase("text"))
    result = new ToTextStream();
result.setEncoding(_encoding);
```

Figure 5. Doomed situations: missing else clause

developer to complain about a null parameter, but no action is taken and thus a null pointer exception will occur. Perhaps null is never provided as an argument to this method. But even if it is, it seems likely that the appropriate remedy for this warning would be to throw a null pointer exception when the parameter is null. Since the existing code already gives this behavior, changing the code is probably unwarranted (although documenting the fact that the parameter must be non-null would be useful).

Figure 5 shows what is effectively a switch statement, constructed using `if .. else` statements. This pattern is relevantly common, even to the detail of not having an else clause for the final `if` statement. Thus, if the final `if` statement fails, `result` will be null and a null pointer exception will occur. While this code is highly questionable, the appropriate fix would likely be to throw an IllegalArgumentException if none of the `if` guards match, and the impact of a null pointer exception is unlikely to be significantly different than that of throwing an IllegalArgumentException.

Figure 6 shows an example where the program has detected an erroneous situation, and is in the process of creating an exception to throw. However, due to a programming error, a null pointer exception will occur when `node` is dereferenced. While the code is clearly mistaken, the impact of the mistake is minimal.

3.5 Testing code

In testing code, developers will often do things that seem nonsensical, such as checking that invoking `.equals(null)` returns false. In this case, the test is checking that the `equals` method can handle a null argument. We can't ignore nonsensical code in testing code, since it may reflect a coding mistake that results in the test not testing what was intended.

```
// com.sun.org.apache.xerces.internal.util
// lines 78-122, abridged
Node node = null;
switch(place.getNodeType()) {
case Node.CDATA_SECTION_NODE: {
  node = ...
  break;
}
case Node.COMMENT_NODE:
...
default: {
  throw new IllegalArgumentException("...("
    + node.getNodeName()+')');
}
```

Figure 6. Doomed situation: error in error handling

```
// com.sun.org.apache.xml.internal.resolver.Catalog
// lines 818-820
String userdir = System.getProperty("user.dir");
userdir.replace('\\', '/');
catalogManager.debug.message(1,
           "Malformed URL on cwd", userdir);
```

Figure 7. Logging bug

3.6 Logging or other unimportant case

We have also seen a number of cases of a bug that would only impact logging output, or assertions. While accurate logging messages are important, bugs in logging code might be deemed to be of lower importance. Figure 7 shows code in which the call to `replace` is performed incorrectly. The `replace` method cannot modify the String it is invoked on - Java Strings are immutable. Rather, it returns a new String that is the result of the modification. Since the return result is ignored here, the call to replace has no effect and the userdir may contain back slashes rather than the intended forward slashes.

3.7 When should such defects be fixed?

Should a defect that doesn't cause the program to significantly misbehave be fixed? The main arguments against fixing such defects is that they require engineering resources that could be better applied elsewhere, and that there is a chance that the attempt to fix the defect will introduce another, more serious bug that *does* significantly impact the behavior of the application. The primary argument for fixing such defects is that it makes the code easier to understand and maintain, and less likely to break in the face of future modifications or uses.

When sophisticated analysis finds an interprocedural error path involving aliasing and multiple conditions, understanding the defect and how and where to remedy it can take significantly more engineering time, and it can be more difficult to have confidence that the remedy resolves the issue without introducing new problems. However, most the warnings generated by FindBugs are fairly obvious once you know what to look for, can be understood by looking at a few lines of code, and the fix is straight forward and obvious. Thus, with FindBugs warnings it is often possible to just understand the defect and fix it without expending the effort required to do a full analysis of the possible impact of the fault (or the fix) on application behavior. However, even simple defects suggest holes in test

Pattern	Warnings	Bad Analysis	Trivial	Impact	Serious
Total	379	5	160	176	38
Nullcheck of value previously dereferenced	54	0	32	20	2
Possible null pointer dereference	48	2	14	28	4
Unwritten field	47	2	35	8	2
Invocation of toString on an array	27	0	0	27	0
Class defines field that masks a superclass field	22	1	9	11	1
Method call passes null for unconditionally dereferenced parameter	17	0	6	9	2
Possible null pointer dereference in method on exception path	16	0	14	2	0
Method ignores return value	13	0	7	4	2
Field only ever set to null	10	0	4	6	0
Suspicious reference comparison	10	0	0	10	0
Read of unwritten field	9	0	9	0	0
A parameter is dead upon entry to a method but overwritten	9	0	2	7	0
Uninitialized read of field in constructor	8	0	7	1	0
Uncallable method defined in anonymous class	8	0	0	8	0
Integer shift by an amount not in the range 0..31	8	0	0	0	8
Doomed test for equality to NaN.	8	0	1	6	1
Call to equals() comparing different types	6	0	0	2	4
Potentially dangerous use of non-short-circuit logic	5	0	0	5	0
Null value is guaranteed to be dereferenced	5	0	4	1	0
Null pointer dereference	5	0	0	2	3
Bad comparison of signed byte	4	0	2	2	0
Call to equals() with null argument	4	0	2	2	0
Impossible cast	3	0	2	1	0
Dead store due to switch statement fall through	3	0	0	0	3
Self assignment of field	3	0	2	1	0
Double assignment of field	3	0	2	0	1
A known null value is checked to see if it is an instance of a type	3	0	1	2	0
int value cast to double and then passed to Math.ceil	3	0	0	3	0
instanceof will always return false	3	0	3	0	0
12 more, less than 3 each	15	0	2	6	7

Table 1. FindBugs medium/high priority "Correctness" warnings on JDK1.6.0-b105

coverage and additional unit tests should be created to supplement defect fixes.

4. Results on Sun's JDK1.6.0

In this section, we present results on Sun's JDK 1.6.0 implementation. We analyzed all 89 publicly available builds of JDK, builds b12 through b105. We only report on the high and medium priority correctness warnings generated by a developmental build of FindBugs 1.1.4. Note that in computing lines of code, we use the number of non-commenting source statements, rather than the number of lines in the file. We can calculate this very accurately from the line number tables associated with each method in a class file. Note that statements spanning multiple lines are counted once this way, as are multiple statements on a single line. This gives a value that is typically 1/3 to 1/4 of the total number of text lines in the file. To compare these results against defect densities based on total number of text lines in the file, you will need to divide our defect densities by 3-4.

Build	b12	b51	b105
# Warnings	370	449	407
warnings/KLocNCSS	0.46	0.45	0.42

These numbers and densities are typical and our target range for applying static analysis to large established code bases. Developers tasked with reviewing a million line code base would throw up their hands in dismay if you presented them with 5,000 issues to review.

We manually examined the defects that were removed during the development of JDK 1.6.0. More precisely, we looked at each warning that was present in one build and not reported in the next version, but the class containing the warning was still present. To simplify the analysis, we only examined defects from files that were distributed with the JDK. Of a total of 53 defect removals, 37 were due to a small targeted program change that seemed to be narrowly focused on remedying the issue described by the warning. Five were program changes that partially but not entirely remedy the issue raised by the warning (the warning was that a null pointer dereference could occur in the computation of a String that was passed to a logging method; the change was to only compute those Strings if logging was enabled, which lowered the issue below a priority that FindBugs reports). The remaining 11 warnings disappeared due to substantial changes or refactorings that had a larger scope than the removal of the one defect.

In Table 1, we report on a manually evaluation of all of the medium and high priority correctness warnings in build 105 of JDK1.6.0 (the official release). We reviewed bugs located in any source file contained in the Java Research License distribution of the JDK, which is more than the source files distributed with the JDK. The bug patterns are described at

`http://findbugs.sourceforge.net/bugDescriptions.html`

Of 379 medium and high priority correctness warnings, we classified:

- 5 as being due to bad analysis by FindBugs (in one case, due to not understanding that a method call could change a field)
- 160 as not being possible or likely to have little or no functional impact for one or more of the reasons described in Section 3
- 176 as seeming to have functional impact, and

- 38 as likely to have substantial functional impact: the method containing the warning will clearly behave in a way substantially at odds with its intended function.

Clearly, any such classification is open to interpretation, and it is likely than other reviewers would produce slightly different classifications. Also, our assessment of the functional impact may differ from the actual end-user perspective. For example, even if a method is clearly broken, the method might never be called and might not be invokable by user code. However, given the localized nature of many of the bug patterns, we have some confidence in the general soundness our classification.

5. Results on Glassfish v2

The Glassfish project is "open source, production-quality, enterprise software. The main deliverables are an Application Server, the Java EE 5 Reference Implementation, and the Java Persistence API Reference Implementation." Members of the Glassfish project have demonstrated substantial interest in the FindBugs project for over a year, and FindBugs is run against their nightly builds. The results are posted on a web page, and recently results have been emailed to developers.

We analyzed Glassfish v2, builds 09-b33. One thing we checked were defects that were present in one version and not reported in the next build. We restricted our analysis to medium and high priority correctness warnings, ignored defects that disappeared because the file containing them was removed, and only considered files in the Glassfish source distribution. There were a total of 58 bug defect disappearances. Of these, 50 disappeared due to small edits designed to specifically address the issue raised by FindBugs (and 17 of the commit messages mentioned FindBugs), and the other 8 disappeared due to more more complicated edits that addressed more than the issue raised by FindBugs.

We also looked at the medium and high priority correctness warnings in build 33, restricting ourselves to files included in the source distribution. We found a total of 334 such warnings, which corresponds to a defect density of 0.44 defects / KLocNCSS.

6. Experiences at Google

Since summer 2006, we have been running FindBugs against portions of Google's Java code base, manually evaluating warnings, and filing bug reports as deemed appropriate. The results of these efforts for FindBugs medium and high priority correctness warnings are shown in Table 2. These data represent continued sampling of the code base covering several projects in various stages of development over a six month period, using several FindBugs versions as available.

We classified each issue as impossible, trivial, or a defect. Issues marked as impossible are ones that we believe could not be exhibited by the code, whereas trivial issues are ones that we believe might happen but would have minimal adverse impact if they did. For checkers looking for relatively syntactic patterns, many of the impossible bug warnings are due errors in the analysis. For checkers that are context dependent, we determined that many of the impossible defects were impossible by examining a larger context than used by the tool. For example, the most common warning was the Redundant Check for Null pattern, where a variable is checked for null after it was already dereferenced and thus cannot be null. In many cases, we believe this warning may be due to a developer reflexively putting a null check before a dereference of a variable, without trying to understand whether or not it is possible for the variable to be null. A typical example of such a defensive null check is shown in Figure 8. In evaluating this warning, we check to see if the dereference was introduced by a later change than the null check. If so, it is possible that a developer introduced

```
x.getY();
... // several to many lines of code
if (x != null && x.isValid()) { ...
```

Figure 8. Typical defensive null check

the dereference without realizing that the variable can be null. This warning, and its partner, a dereference before a null check, are often fixed even when the null cases cannot happen in practice. Apparently, the risk (potential cost) of going forward with inconsistent assumptions in the code outweighs the known cost and risk of making a small change. We did not observe a clear trend when choosing a fix between moving the check before the dereference and simply removing the check.

Compared with the other studies, the Google data includes more cases where the analysis was faulty or a bug pattern was reporting numerous false positives. We attribute this to the fact that the Google data contains results from a range of FindBugs versions over six months. Lessons learned from many of the undesirable false positives have been used to improve FindBugs and many of these issues are not reported by the latest version of FindBugs.

Over time, we observed that the likelihood of a warning being fixed varies with the length of time the issue has existed in the code. After 6 months of running the tool, the average age of the fixed bugs were 5 months versus 9 months for the open bugs. Based on our interaction with developers, we believe older code is less likely to be fixed for several reasons. First, older code may be obsolete, deprecated, or "going away soon." Also, the original developer may have moved to another project, which generally makes it harder to find someone willing to make a change. Finally, older code is generally more trusted and more thoroughly tested and so a "defect" is less likely to lead to surprises.

7. Related work

We had previously discussed [10] the techniques used by FindBugs to track defect occurrences from one analysis run to the next.

Engler et al. have published numerous papers on static analysis for defect detection, and their SOSP 2001 paper [5] have included some experimental evaluation of defect warnings. Li et al. [7] examined the commit logs and messages to classify the occurrence of bugs characteristics and patterns.

There have been few published works evaluating the defects found by commercial defect detection tools, due to the cost and license agreements associated with those tools. Almossawi, Lim and Sinha [1] provide a third party evaluation of Coverity Prevent and reported an overall evaluation than 13% of the warnings seemed be infeasible and 64% seemed very likely to result in faults. The paper did not touch on the issue of true but low impact defects, but since the tool scans C/C++ code and primarily looks for memory access errors, very few true defects could be considered to have low impacts.

Robert O'Callahan blogged [8] about his experience with Klocwork and Coverity, and noted that many of the defects found did not seem to be as significant as the press releases seemed to make them out to be. Konstantin Boundnik [2] blogged about his experience with Klocwork and Coverity, including some high-level data about defect density and false positive rates.

Rutar et al [9] studied ESC/Java, FindBugs, JLint and PMD. However, this paper did not attempt to evaluate the individual warnings or track the history of warnings over time in a software project. Rather, it studied the correlation and overlap between the different tools. Zitser et al.[13] evaluated several open source static analyzer against known exploitable buffer overflows from open source code.

Bug Pattern	Warnings	Impossible	Trivial	Open	Fixed
Total	1127	193	127	289	518
Nullcheck of value previously dereferenced	222	30	49	52	91
Invoking toString on an array	161	1	8	54	98
Possible null pointer dereference	98	28	10	25	35
Method ignores return value	66	8	1	25	32
Call to equals() comparing different types	49	1	0	10	38
int division result cast to double or float	47	7	7	23	10
Null value is guaranteed to be dereferenced	36	21	1	6	8
Null pointer dereference	33	2	0	7	24
An apparent infinite recursive loop	31	0	0	1	30
Questionable used of non-short-circuit logic	30	6	12	5	7
Bad attempt to compute absolute value of signed 32-bit hashcode	30	1	3	5	21
Read of unwritten field	29	5	0	5	19
Bad attempt to compute absolute value of signed 32-bit random integer	26	5	1	12	8
Non-virtual method call passes null of unconditionally dereferenced parameter	24	20	0	0	4
equals() used to compare array and nonarray	17	0	0	4	13
Uncallable method defined in anonymous class	16	0	9	2	5
Possible null pointer dereference in method on exception path	15	8	3	2	2
Uninitialized read of field in constructor	13	7	2	2	2
Impossible cast	13	0	0	4	9
int value cast to float and then passed to Math.round	13	0	5	6	2
Self assignment of field	12	0	1	6	5
Field only ever set to null	11	2	1	1	7
A parameter is dead upon entry to a method but overwritten	9	4	0	1	4
Class defines field that masks a superclass field	9	0	3	4	2
No relationship between generic parameter and method argument	7	0	0	7	0
Dead store due to switch statement fall through	6	0	0	0	6
Unwritten field	6	1	1	1	3
Invocation of equals() on an array, which is equivalent to ==	6	0	0	2	4
Method may fail to close stream	6	2	0	0	4
Very confusing method names	5	2	2	1	0
On an exception path, value is null and guaranteed to be dereferenced	5	3	0	2	0
Useless control flow on next line	5	1	4	0	0
Integer shift by an amount not in the range 0..31	4	0	0	1	3
Method call passes null for unconditionally dereferenced parameter	4	4	0	0	0
Others - less than 4 each	60	22	4	13	21

Table 2. FindBugs medium/high priority "Correctness" warnings from code at Google

Wagner et al. [11] evaluated FindBugs and PMD on two software projects. They found that very few of the defects identified, post-release, by these tools actually correlated to documented failures in deployed software. Of a total of 91 defect removals, comparing two successive versions of software, they found only 4 that corresponded to remedying a problem that caused a failure in the deployed software.

8. Conclusions

One major point of this paper is to discuss the fact that trying to classify defect warnings into false positives and true positives oversimplifies the issue, and that obviously bogus and inconsistent coding mistakes may have no adverse impact on software functionality. In memory safe programming languages, a substantial portion of defects detected by static analysis might fall into this category. Removing such defects may make software easier to maintain and is still useful. However, the costs associated with such removal must be kept low. Trying to devise static analysis techniques that suppress or deprioritize true defects with minimal impact, and highlight defects with significant impact, is an important and interesting research question.

We believe that one of the reasons that static analysis sometimes reports true but trivial issues is that static analysis tools, in general, don't know what the code is *supposed* to do. Thus, it can't check that the code correctly implements what it is supposed to do. Instead, many static analysis techniques look for unusual code, such as an awkward or dubious computation, or a statement that if executed will result in an exception but might in fact be dead code that can never be executed. These are instances of bad coding, which ideally should be cleaned up, but might not impair the actual functionality of the software. In contrast, testing is usually directed at testing the issues that the developer believed to be important, and thus errors found in testing more frequently correspond to impairments of functionality. Static analysis driven by more focused goals (e.g., SQL injection must not be possible) or developer annotations of intended behavior may provide better ways to identify high-impact issues.

The existence of low impact defects isn't a barrier to adoption of static analysis technologies; a substantial portion of the issues found do have functional impact, and reviewing warnings to discern which have apparent functional impact isn't a problem. However, any serious application of static analysis techniques needs to understand that after the issues with functional impact are resolved, there will be a significant number of issues with little or no functional impact. Some plan needs to be put in place for how to deal with these. In some cases, fixing "ugly, nasty" code might be appro-

priate, even with no apparent functional impact. But in many cases, the code won't be changed, and mechanisms must be in place so these issues do not need to be reviewed again each time the code is analyzed.

We have also shown that FindBugs reports a substantial number of defects on production software, and while several bug patterns account for a majority of the defects, that there is a "long tail" to bug patterns: lots of bug patterns, each of which might typically find only a few defects per million lines of code. FindBugs reports more than 250 different bug patterns, over a hundred of which are classified as correctness bugs. This is feasible because many of the bug patterns are small and focused, and can be initially implemented and evaluated within an hour or two.

We've also shown that the defects reported by FindBugs are issues that developers are willing to address. In the case of the Sun's JDK, no systematic policy or procedure of reviewing FindBugs warnings was in place, yet the build history shows developers specifically making changes to address the issues raised by FindBugs. While some of changes might have been informed by FindBugs, many of the changes occurred over a year ago, when FindBugs was in a much more primitive state, not widely adopted, and did not, at the time, report many of the defects which were removed.

The experience in the Glassfish and at Google show a more systematic effort to address defects found through static analysis. The Google effort has been particularly well focused, and incorporates three elements that we believe are important:

- a centralized effort to run and perform an initial triage of warnings, so that individual developers are notified, without any action on their part, of issues in their code, and
- an integrated system to be able to mark defects connecting the results of previous triages with the results of new analysis, so that defects marked as not being issues don't have to be reexamined after each analysis, and
- a system for reviewing the effectiveness of the static analysis, to allow for determination of effective bug patterns, feedback to static analysis tool vendors, and review of the effectiveness and impact of the static analysis effort.

At Google, nearly all instances of the bug patterns identified as generally important have been triaged, and new warnings, whether resulting from new check-ins or new versions of the analysis, are initially reviewed within days.

The feedback from the team performing bug triage at Google has been very helpful in improving the accuracy, classification and prioritization of warnings reported by FindBugs.

The effort on the Glassfish has resulted in a number of issues being examined and addressed, but FindBugs identifies 341 high and medium priority correctness defects in build b33. Some of the remaining 341 warnings are undoubtedly trivial or not possible, but clearly some issues with functional impact remain, and there is no central location for viewing any comments as to the significance or insignificance or any particular warning.

Google confidentiality requirements prevent any direct quantitative comparison of defects in the Sun code base and the Google code base. However, UMD has scanned a number of open code bases and closed code bases, including those in commercial closed source Java applications that can be downloaded over the web. In looking at defects found in many systems, including the JDK, Glassfish, IBM WebSphere, JBoss, BEA WebLogic, Oracle Containers for Java, SleepyCat Java DB, we have generally seen defect densities of 0.3 - 0.6 warnings/KLocNCSS (medium and high priority correctness warnings per 1,000 lines of non-commenting source statements). The only times we have see defect densities significantly lower than that is when, as in SleepyCat DB, there has been a systematic attempt to review and address issues raised by static analysis.

9. Acknowledgments

Thanks to David Hovemeyer for his comments on the paper, and to all those who have contributed to the FindBugs project. Fortify Software is the sponsor of the FindBugs project, and additional current support is provided by Google and Sun Microsystems. Sandra Cruze, Antoine Picard, Shama Butala, Boris Debic and Feng Qian helped us successfully deploy FindBugs at Google.

References

[1] A. Almossawi, K. Lim, and T. Sinha. Analysis tool evaluation: Coverity prevent, May 2006. http://www.cs.cmu.edu/ aldrich/courses/654/tools/cure-coverity-06.pdf.

[2] K. I. Boudnik. Static analyzers comparison, October 2006. http://weblogs.java.net/blog/cos/archive/2006/10/static_analyzer.html.

[3] B. Chelf, D. Engler, and S. Hallem. How to write system-specific, static checkers in metal. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 51–60, New York, NY, USA, 2002. ACM Press.

[4] T. Copeland. *PMD Applied*. Centennial Books, November 2005.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[6] T. D. Isil Dillig and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2007.

[7] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, New York, NY, USA, 2006. ACM Press.

[8] R. O'Callahan. Static analysis and scary headlines, September 2006. http://weblogs.mozillazine.org/roc/archives/2006/09/static_analysis_and_scary_head.html.

[9] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[10] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136, New York, NY, USA, 2006. ACM Press.

[11] S. Wagner, F. Deissenboeck, M. A. J. Wimmer, and M. Schwalb. An evaluation of bug pattern tools for java, January 2007. *unpublished*.

[12] S. Wagner, J. Jurjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Proc. 17th International Conference on Testing of Communicating Systems*, pages 40–55, 2005.

[13] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, New York, NY, USA, 2004. ACM Press.